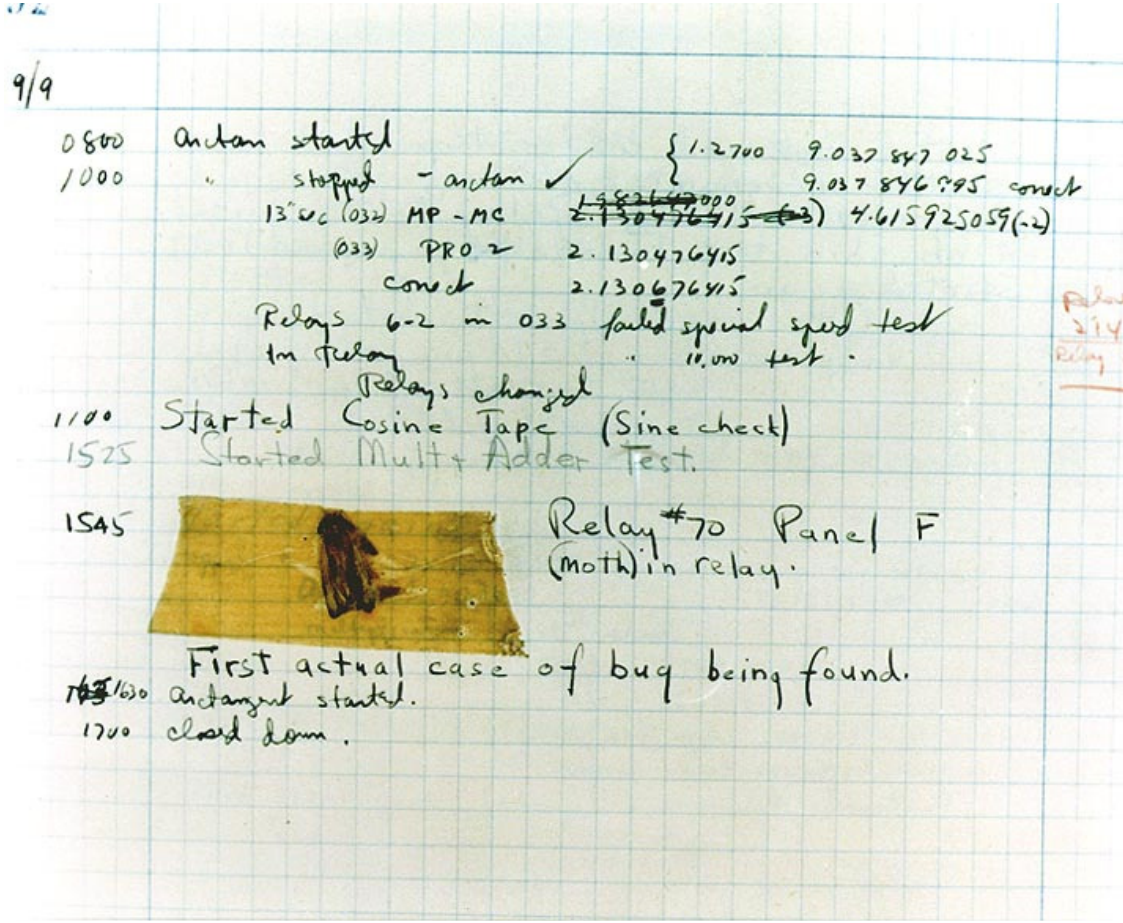
Unit test

…Debugging

While Grace Hopper was working on the Harvard Mark II Computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system.



Though the term computer bug cannot be definitively attributed to Admiral Hopper, she did bring the term into popularity.

The remains of the moth can be found in the group's log book at the Smithsonian Institution's National Museum of American History in Washington, D.C.

Motivations

- As most software tend to grow in complexity over time, it becomes increasingly difficult to test. Less complicated software can be manually tested by pushing buttons in a structured way, but at a certain point the testing task becomes too complex for a manual test and only a fraction of the required tests are actually executed properly.
- If no action is taken quality will decrease. Eventually the software will end in a state where adding one error correction introduces several other errors.
- As the software itself needs to be broken into more manageable units, so do the tests. As the new units are developed, it makes sense to consider adding automated unit tests, where each unit is isolated and tested out of its context.
- The tests needs to be adapted to new features and functionality in the software.

Components of a unit test suite

- Need to setup/teardown the test environment before/after the execution of the test suite
- Need to manage a test registry, suites, and test cases setting up an appropriate library
- Need to mock/replace any symbols external to the module being tested. Mocking/replacement is done with functions that return values determined by the test - within the test application
- Need to check correctness of values returned by the test (assert correctness)
- Need to view results (reporting)
- Need to protect test environment (be able to restore test environment if one of the test fails)

What test process are we referring to ? (Extreme programming?)

Process

- Write tests before writing the code

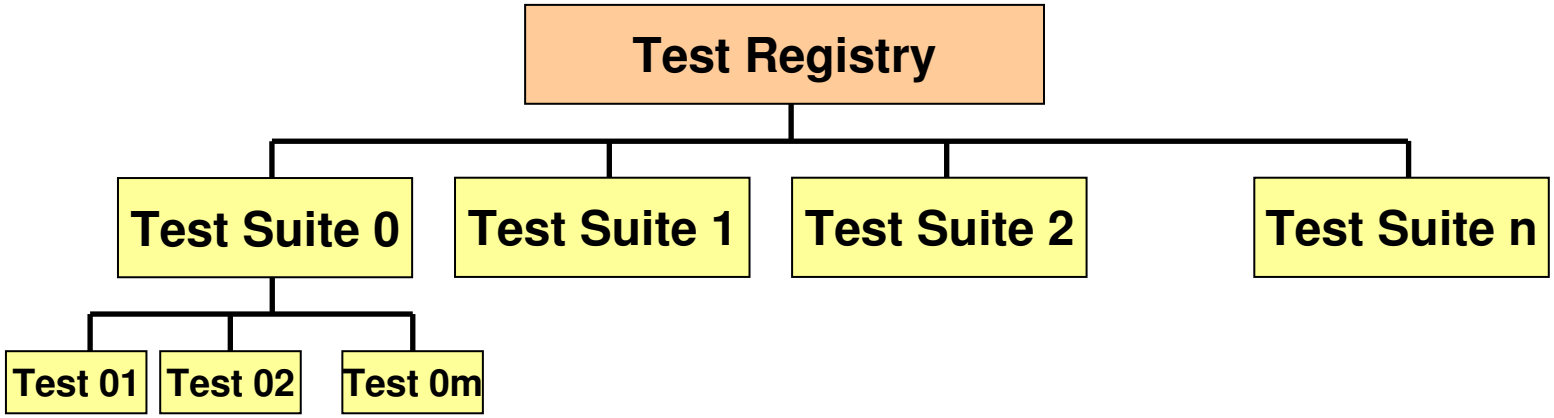
CUnit

http://cunit.sourceforge.net/

- Strenghts
  - Platform independent
  - Simple (a static library linked with the user code)
  - Some reporting (txt + XML)
  - Lots of assertion capability
  - Good management of Test registry and suites.
- Weaknesses
  - Little support for mockery
  - No protection of test environment

CUnit

- The core framework provides basic support for managing a test registry, suites, and test cases.
- The user interfaces facilitate interaction with the framework to run tests and view results.



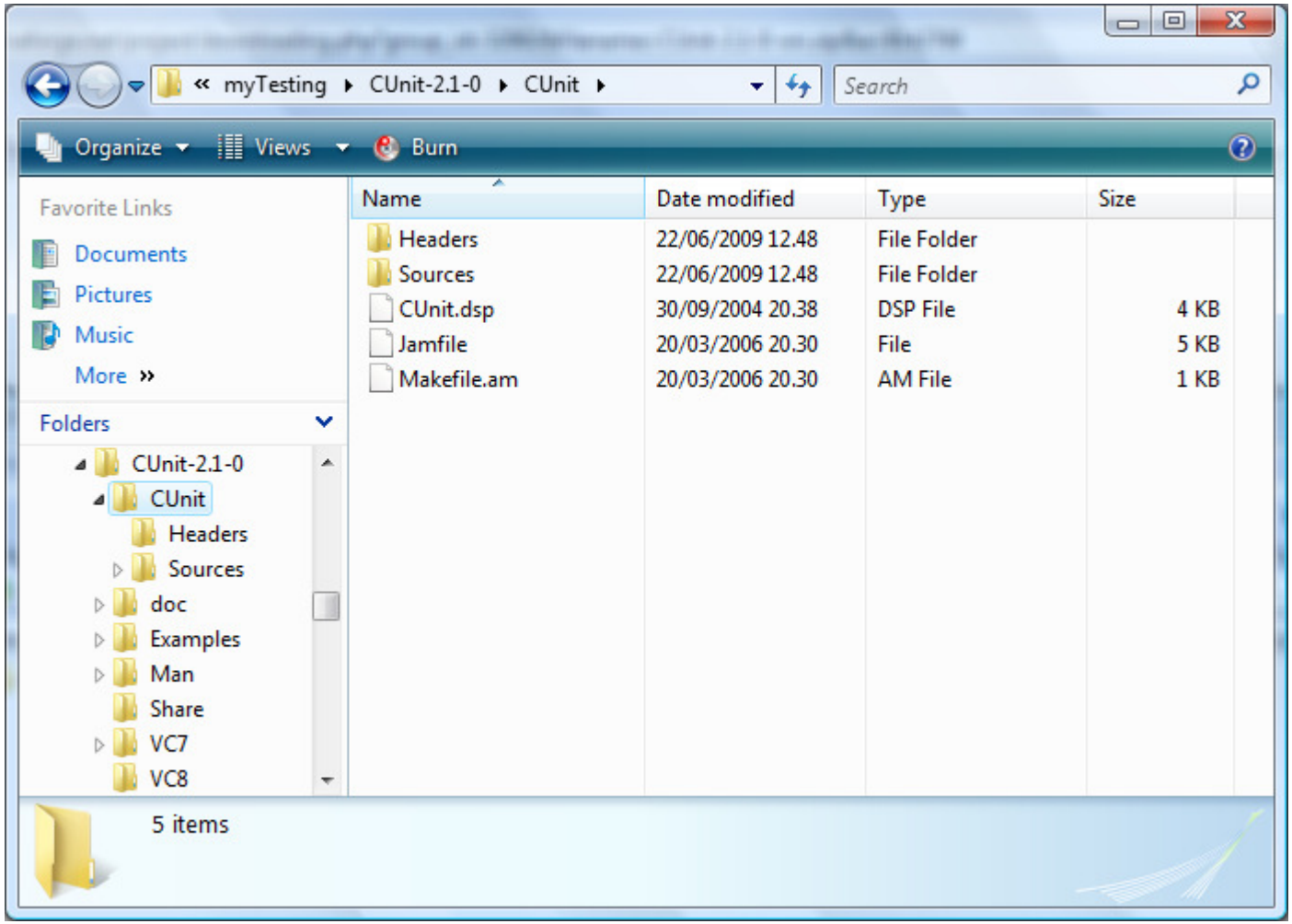
- Test cases are packaged into suites, which are registered with the active test registry.
- Suites can have setup and teardown functions which are automatically called before and after running the suite's tests.
- All suites/tests in the registry may be run using a single function call, or selected suites/tests can be run.

CUnit

- A typical sequence of steps for using the CUnit framework is:
  1. Write functions for tests (and suite init/cleanup if necessary).
  2. Initialize the test registry - CU_initialize_registry()
  3. Add suites to the test registry - CU_add_suite()
  4. Add tests to the suites - CU_add_test()
  5. Run tests using an appropriate interface, e.g. CU_console_run_tests
  6. Cleanup the test registry - CU_cleanup_registry

CUnit

- Contents



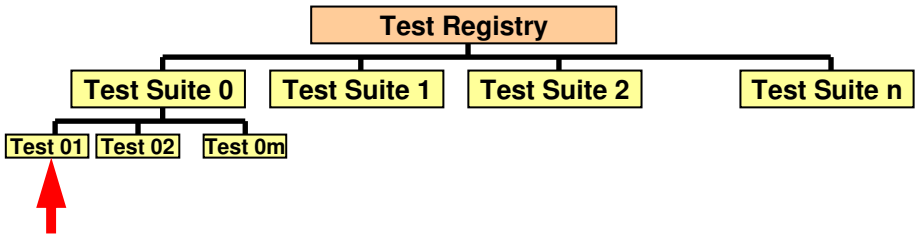
CUnit

- Contents …

Header File	Description
#include <CUnit/CUnit.h>	ASSERT macros for use in test cases, and includes other framework headers.
#include <CUnit/CUError.h>	Error handing functions and data types. Included automatically by CUnit.h.
#include <CUnit/TestDB.h>	Data type definitions and manipulation functions for the test registry, suites, and tests. Included automatically by CUnit.h.
#include <CUnit/TestRun.h>	Data type definitions and functions for running tests and retrieving results. Included automatically by CUnit.h.
#include <CUnit/Automated.h>	Automated interface with xml output.
#include <CUnit/Basic.h>	Basic interf. w. non-interactive output to stdout.
#include <CUnit/Console.h>	Interactive console interface.

CUnit

- Defining tests as pairs
  - the function(s) to be tested
  - The test function
- Example




```
int maxi(int i1, int i2)
{
  return (i1 > i2) ? i1 : i2;
}
```

```
void test_maxi(void)
{
  CU_ASSERT(maxi(0,2) == 2);
  CU_ASSERT(maxi(0,-2) == 0);
  CU_ASSERT(maxi(2,2) == 2);
}
```

A test function runs only after having been registered in a test suite.

CUnit

```
int maxi(int i1, int i2)
{
  return (i1 > i2) ? i1 : i2;
}
```

```
void test_maxi(void)
{
  CU_ASSERT(maxi(0,2) == 2);
  CU_ASSERT(maxi(0,-2) == 0);
  CU_ASSERT(maxi(2,2) == 2);
}
```

Assertions

- CUnit provides a set of *assertions* for testing logical conditions. The success or failure of these assertions is tracked by the framework, and can be viewed when a test run is complete.
- Each assertion tests a single logical condition, and fails if the condition evaluates to FALSE. Upon failure, the test function continues unless the user chooses the 'xxx_FATAL' version of an assertion. In that case, the test function is aborted and returns immediately.

CUnit

```
void test_longjmp(void)
{
  jmp_buf buf;
  int i;

  i = setjmp(buf);
  if (i == 0) {
    run_other_func();
    CU_PASS("run_other_func() succeeded.");
  } else
    CU_FAIL("run_other_func() issued longjmp.");
}
```

Assertions

- There are also special "assertions" (PASS and FAIL) for registering a pass or fail with the framework without performing a test. These are useful for testing flow of control or other conditions not requiring a logical test

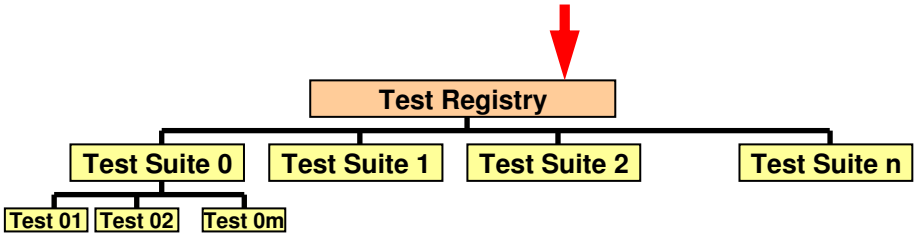
CUnit

Available Assertions

CU_ASSERT(int expression) **CU_ASSERT_FATAL**(int expression) **CU_TEST**(int expression) **CU_TEST_FATAL**(int expression)	Assert that *expression* is `TRUE` (non-zero)
CU_ASSERT_TRUE(value) **CU_ASSERT_TRUE_FATAL**(value)	Assert that *value* is `TRUE (non-zero)`
CU_ASSERT_FALSE(value) **CU_ASSERT_FALSE_FATAL**(value)	Assert that *value* is `FALSE` (zero)
CU_ASSERT_EQUAL(actual, expected) **CU_ASSERT_EQUAL_FATAL**(actual, expected)	Assert that *actual* = = *expected*
CU_ASSERT_NOT_EQUAL(actual, expected)) **CU_ASSERT_NOT_EQUAL_FATAL**(actual, expected)	Assert that *actual* != *expected*
CU_ASSERT_PTR_EQUAL(actual, expected) **CU_ASSERT_PTR_EQUAL_FATAL**(actual, expected)	Assert that pointers *actual* = = *expected*
CU_ASSERT_PTR_NOT_EQUAL(actual, expected) **CU_ASSERT_PTR_NOT_EQUAL_FATAL**(actual, expected)	Assert that pointers *actual* != *expected*
CU_ASSERT_PTR_NULL(value) **CU_ASSERT_PTR_NULL_FATAL**(value)	Assert that pointer *value* == NULL
CU_ASSERT_PTR_NOT_NULL(value) **CU_ASSERT_PTR_NOT_NULL_FATAL**(value)	Assert that pointer *value* != NULL
CU_ASSERT_STRING_EQUAL(actual, expected) **CU_ASSERT_STRING_EQUAL_FATAL**(actual, expected)	Assert that strings *actual* and *expected* are equivalent
CU_ASSERT_STRING_NOT_EQUAL(actual, expected) **CU_ASSERT_STRING_NOT_EQUAL_FATAL**(actual, expected)	Assert that strings *actual* and *expected* differ
CU_ASSERT_NSTRING_EQUAL(actual, expected, count) **CU_ASSERT_NSTRING_EQUAL_FATAL**(actual, expected, count)	Assert that 1st count chars of *actual* and *expected* are the same
CU_ASSERT_NSTRING_NOT_EQUAL(actual, expected, count) **CU_ASSERT_NSTRING_NOT_EQUAL_FATAL**(actual, expected, count)	Assert that 1st count chars of *actual* and *expected* differ
CU_ASSERT_DOUBLE_EQUAL(actual, expected, granularity) **CU_ASSERT_DOUBLE_EQUAL_FATAL**(actual, expected, granularity)	Assert that |*actual* - *expected*| <= |*granularity*| *Math library must be linked in for this assertion.*
CU_ASSERT_DOUBLE_NOT_EQUAL(actual, expected, granularity) **CU_ASSERT_DOUBLE_NOT_EQUAL_FATAL**(actual, expected, granularity)	Assert that |*actual* - *expected*| > |*granularity*| *Math library must be linked in for this assertion.*
CU_PASS(message)	Register a passing assertion with the specified message. No logical test is performed.
CU_FAIL(message) **CU_FAIL_FATAL**(message)	Register a failed assertion with the specified message. No logical test is performed.

CUnit

Test Registry



The Test registry is the repository for suites and associated tests.

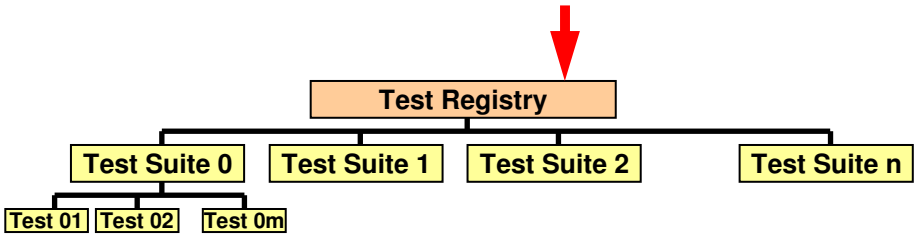
CUnit maintains an active test registry which is updated when the user adds a suite or test. The suites in this active registry are run when the user chooses to run *all tests*.

The CUnit test registry is a data structure CU_TestRegistry declared in <CUnit/TestDB.h>. It includes fields for the total numbers of suites and tests stored in the registry, as well as a pointer to the head of the linked list of registered suites.

```
typedef struct CU_TestRegistry
{
  unsigned int uiNumberOfSuites;
  unsigned int uiNumberOfTests;
  CU_pSuite    pSuite;
} CU_TestRegistry;

typedef CU_TestRegistry* CU_pTestRegistry;
```

CUnit



The user normally only needs to initialize the registry before use and clean up afterwards.

Initialization

`CU_ErrorCode CU_initialize_registry(void)`

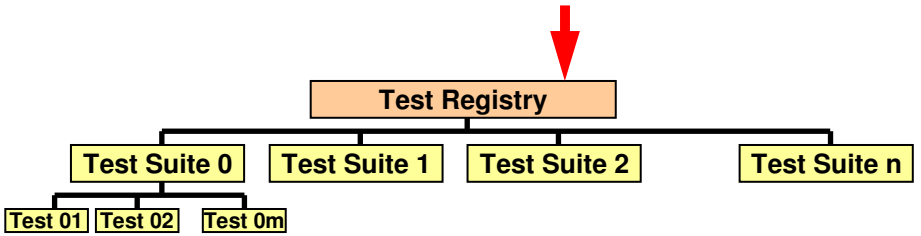
The active CUnit test registry must be initialized before use. The user should call `CU_initialize_registry()` before calling any other CUnit functions. Failure to do so will likely result in a crash.

Cleanup

`void CU_cleanup_registry(void)`

When testing is complete, the user should call this function to clean up and release memory used by the framework. This should be the last CUnit function called. Failure to call *`CU_cleanup_registry()`* will result in memory leaks. It may be called more than once without creating an error condition. *Note that this function will destroy all suites (and associated tests) in the registry*.

CUnit



Other functions are provided to manipulate the registry when necessary.

`CU_pTestRegistry CU_get_registry(void)`

Returns a pointer to the active test registry.

`CU_pTestRegistry CU_set_registry(CU_pTestRegistry pTestRegistry)`

Replaces the active registry with the specified one. A pointer to the previous registry is returned. *It is the caller's responsibility to destroy the old registry*.

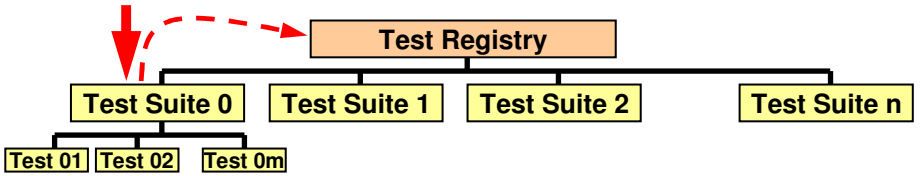
`CU_pTestRegistry CU_create_new_registry(void)`

Creates a new registry and returns a pointer to it. The new registry will not contain any suites or tests. It is the caller's responsibility to destroy the new registry by one of the mechanisms described previously.

`void CU_destroy_existing_registry(CU_pTestRegistry* ppRegistry)`

Destroys and frees all memory for the specified test registry, including any registered suites and tests.

CUnit



Managing Suites

The test collection (suite) must be registered with the test registry.

```
CU_pSuite CU_add_suite(const char* strName, CU_InitializeFunc pInit,
                                            CU_CleanupFunc pClean)
```

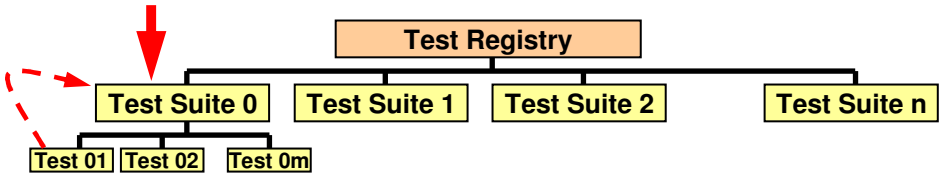
Creates a new test collection (suite) having the specified name, initialization function, and cleanup function. The new suite is registered with (and owned by) the test registry

- The registry must be initialized before adding any suites.

A pointer to the new suite is returned, which is needed for adding tests.

The suite's name must be unique. The initialization and cleanup functions are optional, passed as pointers to functions to be called before and after the tests in the suite. This allows to set up and tear down fixtures to support the tests. These functions take no arguments and should return zero if they are completed successfully (non-zero otherwise). If a suite does not require one or both of these functions, pass NULL.

CUnit



Managing Suites

In order for a test to be run by CUnit, it must be added to a test suite.

```
CU_pTest CU_add_test(CU_pSuite pSuite, const char* strName,
                     CU_TestFunc pTestFunc)
```

Creates a new test having the specified name and test function, and registers it with the specified suite. The suite must already have been created using `CU_add_suite()`.

The test's name must be unique among all tests added to a single suite. The test function cannot be NULL, and points to a function to be called when the test is run. Test functions have neither arguments nor return values.

A pointer to the new test is returned. If an error occurs during creation of the test, NULL is returned and the framework error code is set

CUnit

Running tests

CUnit supports running all tests in all registered suites, but individual tests or suites can also be run.
During each run, the framework keeps track of the number of suites, tests, and assertions run, passed, and failed. The results are cleared each time a test run is initiated (even if it fails).
Simplified user interfaces handle the details of interaction with the framework and provide output of test details and results for the user.
The following interfaces are included in the CUnit library:

Automated: non-interactive with output to xml files
Basic: non-interactive with optional output to stdout
Console: interactive console mode under user control

CUnit

Automated Mode

In this mode clients initiate a test run, and the results are output to an XML file. A listing of the registered tests and suites can also be reported.

`void` **`CU_automated_run_tests`**`(void)`

Runs all tests in all registered suites. Test results are output to a file named *ROOT-Results.xml*. The filename *ROOT* can be set using `CU_set_output_filename()`, or CUnitAutomated-Results.xml is used.

- A DTD file *CUnit-Run.dtd* and XSL stylesheet *CUnit-Run.xsl*. are provided in the *Share* subdirectory of the source and installation trees.

`CU_ErrorCode` **`CU_list_tests_to_file`**`(void)`

Lists the registered suites and associated tests to file. The listing file is named *ROOT-Listing.xml*. The default for ROOT is *CUnitAutomated*.

- The listing file is supported by both a document type definition file (*CUnit-List.dtd*) and XSL stylesheet (*CUnit-List.xsl*).
- A listing file is not generated automatically by CU_automated_run_tests().

`void` **`CU_set_output_filename`**`(const char* szFilenameRoot)`

Sets the output filenames for the results and listing files.

CUnit

Basic Mode

In basic mode, results are output to stdout. This interface supports running individual suites or tests, and allows client code to control the type of output.

`CU_ErrorCode CU_basic_run_tests(void)`

Runs all tests in all registered suites. Returns the 1st error code during the test run. The type of output is controlled by the current run mode.

`CU_ErrorCode CU_basic_run_suite(CU_pSuite pSuite)`

Runs all tests in single specified suite. Returns the 1st error code ....

`CU_ErrorCode CU_basic_run_test(CU_pSuite pSuite, CU_pTest pTest)`

Runs a single test in a specified suite. Returns the 1st error code ...

`void CU_basic_set_mode(CU_BasicRunMode mode)`

Sets the basic run mode, which controls the output during test runs.

Choices are: *CU_BRM_NORMAL* Failures and run summary are printed.
CU_BRM_SILENT No output is printed except error messages.
CU_BRM_VERBOSE Maximum output of run details.

CUnit

Basic Mode

`CU_BasicRunMode` **`CU_basic_get_mode`**`(void)`

Retrieves the current basic run mode code.

`void` **`CU_basic_show_failures`**`(CU_pFailureRecord pFailure)`

Prints a summary of all failures to stdout.

CUnit

Interactive Console Mode

The console interface is interactive. All the client needs to do is initiate the console session, and the user controls the test run interactively. This include selection & running of registered suites and tests, and viewing test results. To start a console session, use *void* ***CU_console_run_tests****(void)*

Getting Test Results

The interfaces present results of test runs, but client code may sometimes need to access the results directly. These results include various run counts, as well as a linked list of failure records holding the failure details. Results are overwritten each time a new test run is started, or when the registry is initialized or cleaned up. Functions for accessing the test results are:

unsigned int ***CU_get_number_of_suites_run****(void)*
unsigned int ***CU_get_number_of_suites_failed****(void)*
unsigned int ***CU_get_number_of_tests_run****(void)*
unsigned int ***CU_get_number_of_tests_failed****(void)*
unsigned int ***CU_get_number_of_asserts****(void)*
unsigned int ***CU_get_number_of_successes****(void)*
unsigned int ***CU_get_number_of_failures****(void)*

These functions report the number of suites, tests, and assertions that ran or failed during the last run. A suite is considered failed if it's initialization or cleanup function returned non-NULL. A test fails if any of its assertions fail.

CUnit

Error Handling

Most CUnit functions set an error code indicating the framework error status. Some functions return the code, while others just set the code and return some other value.

Two functions are provided for examining the framework error status:

```
CU_ErrorCode CU_get_error(void)
const char* CU_get_error_msg(void)
```

The first returns the error code itself, while the second returns a message describing the error status. The error code is an enum of type `CU_ErrorCode` defined in <CUnit/CUError.h>.

CUnit

Behavior Upon Framework Errors

The default behavior when an error condition is encountered is for the error code to be set and execution continued. There may be times when clients prefer for a test run to stop on a framework error, or even for the test application to exit. This behavior can be set by the user, for which the following functions are provided:

```
void CU_set_error_action(CU_ErrorAction action)
CU_ErrorAction CU_get_error_action(void)
```

The error action code is an enum of type `CU_ErrorAction` defined in <CUnit/CUError.h>. The following error action codes are defined:

- *CUEA_IGNORE* Runs should be continued when an error condition occurs (default)
- *CUEA_FAIL* Runs should be stopped when an error condition occurs
- *CUEA_ABORT* The application should exit() when an error conditions occurs

CMockery

http://cmockery.googlecode.com/

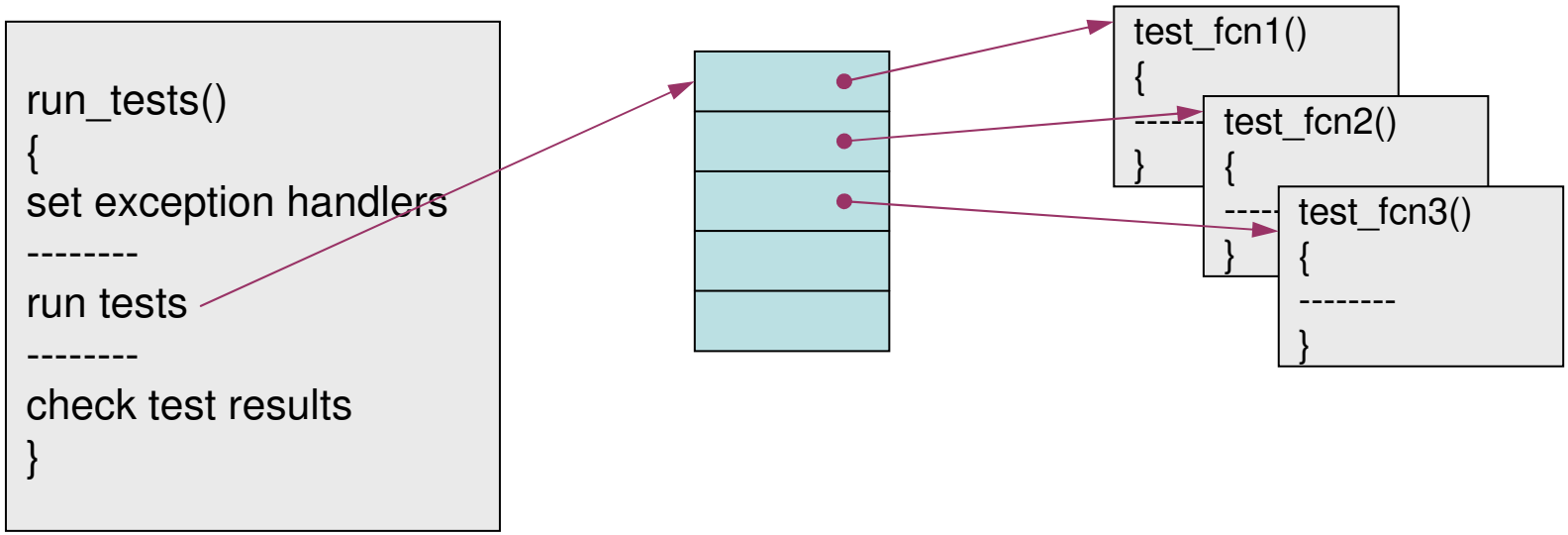
- Strenghts
  - Platform independent
  - Google project
  - Simple (only requires standard C library)
  - Good management of mockery.
- Weaknesses
  - Simple (table based) definition of suite

CMockery

- Cmockery only requires a test application is linked with the standard C library
- Cmockery tests are compiled into stand-alone executables and linked with the Cmockery library, the standard C library and module being tested.
- Any symbols external to the module being tested should be mocked - replaced with functions that return values determined by the test - within the test application.
- It may not be possible to compile a module into a test application without some modification, therefore the preprocessor symbol **UNIT_TESTING** should be defined when Cmockery unit test applications are compiled so code within the module can be conditionally compiled for tests.

CMockery: tests

- Cmockery unit test cases are functions with the signature **void function(void **state)**.
- Cmockery test applications initialize a table with test case function pointers using **unit_test*()** macros.
- This table is then passed to the **run_tests()** macro to execute the tests.
- **run_tests()** sets up the appropriate exception / signal handlers and other data structures prior to running each test function.
- When a unit test is complete **run_tests()** performs various checks to determine whether the test succeeded.



CMockery: tests

Cmockery test applications initialize a table with test case function pointers using **unit_test*()** macros.

```
// Initializes a UnitTest structure.
#define unit_test(f) { #f, f, UNIT_TEST_FUNCTION_TYPE_TEST }
```

“function_name”	function pointer	OP CODE for function

```
#define unit_test_setup(test, setup) \
{ #test "_" #setup, setup, UNIT_TEST_FUNCTION_TYPE_SETUP }

#define unit_test_teardown(test, teardown) \
{ #test "_" #teardown, teardown,
UNIT_TEST_FUNCTION_TYPE_TEARDOWN }
```

Example of use

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>
// A test case that does nothing and succeeds.
void null_test_success(void **state) {
}

int main(int argc, char* argv[]) {
    const UnitTest tests[] = {
        unit_test(null_test_success),
    };
    return run_tests(tests);
}
```

Exception handling

- Before a test function is executed by **run_tests()**, exception / signal handlers are overridden with a handler that simply displays an error and exits a test function.
- The test function is aborted and the application's execution resumes with the next test function.
- Test failures are ultimately signalled via the Cmockery function fail(). The following events will result in a test failure...
  - Assertions
  - Exceptions
  - Memory leaks
  - Mismatched setup and tear down functions
  - Missing mock return values
  - Unused mock return values
  - Missing expected parameter values
  - Unused expected parameter values

Assertions

- Cmockery defines the mock_assert() function.
- mock_assert() signals a test failure.
- If a function is called using the expect_assert_failure() macro, a test failure is signalled if no calls to mock_assert() occur during the function called via expect_assert_failure().

Assertions

Example of mock_assert()

```
#include <assert.h>
// override assert with mock_assert().
#if UNIT_TESTING
extern void mock_assert(const int result, const char* const
            expression, const char * const file, const int line);
#undef assert
#define assert(expression) \
mock_assert((int)(expression), #expression, __FILE__, __LINE__);
#endif // UNIT_TESTING

void increment_value(int * const value) {
  assert(value);
  (*value) ++;
}
void decrement_value(int * const value) {
  if (value) { *value --; }
}
```

Assertions

Macros

- Cmockery provides an assortment of assert macros that tests applications should use use in preference to the C standard library's assert macro.
- On an assertion failure a Cmockery assert macro will write the failure to the standard error stream and signal a test failure.
- Due to limitations of the C language the general C standard library assert() and Cmockery's assert_true() and assert_false() macros can only display the expression that caused the assert failure.
- Cmockery's type specific assert macros, assert_{type}_equal() and assert_{type}_not_equal(), display the data that caused the assertion failure which increases data visibility aiding debugging of failing test cases.

Assertions macros: example

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmockery.h>
extern const char* get_status_code_string(const unsigned int status_code);
extern unsigned int string_to_status_code( const char* const status_code_string);
/* This test will fail since the string returned by get_status_code_string(0)
 * doesn't match "Connection timed out".
 */
void get_status_code_string_test(void **state) {
  assert_string_equal(get_status_code_string(0), "Address not found");
  assert_string_equal(get_status_code_string(1), "Connection timed out");
}
// This test will fail since the status code of "Connection timed out" isn't 1
void string_to_status_code_test(void **state) {
  assert_int_equal(string_to_status_code("Address not found"), 0);
  assert_int_equal(string_to_status_code("Connection timed out"), 1);
}

int main(int argc, char *argv[]) {
  const UnitTest tests[] = { unit_test(get_status_code_string_test),
                             unit_test(string_to_status_code_test), };
  return run_tests(tests);
}
```

Checks on dynamic memory allocation

These are skipped for now …

Mock functions

- Dependency of the unit from external functions/modules is modeled using mock functions that are either statically or dynamically linked with the module being tested.
- Mock functions must be statically linked when the code being tested directly references external functions. Dynamic linking is simply the process of setting a function pointer in a table used by the tested module to reference a mock function defined in the unit test.

Mock functions

Return Values

- In order to simplify the implementation of mock functions Cmockery provides functionality for storing the return values provided by mock functions in each test case using will_return().
- These values are then returned by each mock function using calls to mock(). Values passed to will_return() are added to a queue for each function specified.
- Each successive call to mock() from a function removes a return value from the queue. This makes it possible for a mock function to use multiple calls to mock() to return output parameters in addition to a return value. In addition this allows the specification of return values for multiple calls to a mock function.

Mock functions

Expected Values

- In addition to storing the return values of mock functions, Cmockery provides functionality to store expected values for mock function parameters using the expect_*() functions provided. A mock function parameter can then be validated using the check_expected() macro.
- Successive calls to expect_*() macros for a parameter queues values to check the specified parameter. check_expected() checks a function parameter against the next value queued using expect_*(), if the parameter check fails a test failure is signalled. In addition if check_expected() is called and no more parameter values are queued a test failure occurs.

Examples for Mock functions

```
extern DatabaseConnection* connect_to_customer_database();
extern unsigned int get_customer_id_by_name(
    DatabaseConnection * const connection, const char * const customer_name);
// Mock query database function.
unsigned int mock_query_database( DatabaseConnection* const connection,
  const char * const query_string, void *** const results) {
  *results = (void**)mock();
  return (unsigned int)mock();
}
// Mock of the connect to database function.
DatabaseConnection* connect_to_database(const char * const database_url, const
    unsigned int port) {
  return (DatabaseConnection*)mock();
}
void test_connect_to_customer_database(void **state) {
  will_return(connect_to_database, 0x0DA7ABA53);
  assert_true(connect_to_customer_database() ==
    (DatabaseConnection*)0x0DA7ABA53);
}
void fail_connect_to_customer_database(void **state) {
  will_return(connect_to_database, 0x0DA7ABA53);
  assert_true(connect_to_customer_database() ==
    (DatabaseConnection*)0x0DA7ABA53);
}
```

Examples for Mock functions

```
/* This test fails as the mock function connect_to_database() will have no
 * value to return.
 */
void fail_connect_to_customer_database(void **state)
{
  will_return(connect_to_database, 0x0DA7ABA53);
  assert_true(connect_to_customer_database() ==
    (DatabaseConnection*)0x0DA7ABA53);
}
void test_get_customer_id_by_name(void **state) {
  DatabaseConnection connection = { "somedatabase.somewhere.com", 12345678,
   mock_query_database };
  // Return a single customer ID when mock_query_database() is called.
  int customer_ids = 543;
  will_return(mock_query_database, &customer_ids);
  will_return(mock_query_database, 1);
  assert_int_equal(get_customer_id_by_name(&connection, "john doe"), 543);
}
int main(int argc, char* argv[]) {
  const UnitTest tests[] = { unit_test(test_connect_to_customer_database),
                             unit_test(fail_connect_to_customer_database),
                             unit_test(test_get_customer_id_by_name), };
  return run_tests(tests);
}
```

Mock functions

Checking Parameters

- In addition to storing the return values of mock functions, Cmockery provides functionality to store expected values for mock function parameters using the expect_*() functions provided. A mock function parameter can then be validated using the check_expected() macro.
- Successive calls to expect_*() macros for a parameter queues values to check the specified parameter. check_expected() checks a function parameter against the next value queued using expect_*(), if the parameter check fails a test failure is signalled. In addition if check_expected() is called and no more parameter values are queued a test failure occurs.

Mock functions

```
extern DatabaseConnection* connect_to_product_database();
/* Mock connect to database function.
 * NOTE: This mock function is very general could be shared between tests
 * that use the imaginary database.h module.
 */
DatabaseConnection* connect_to_database(const char * const url,
       const unsigned int port)
{
  check_expected(url);
  check_expected(port);
  return (DatabaseConnection*)mock();
}
void test_connect_to_product_database(void **state) {
  expect_string(connect_to_database, url, "products.abcd.org");
  expect_value(connect_to_database, port, 322);
  will_return(connect_to_database, 0xDA7ABA53); a
  ssert_int_equal(connect_to_product_database(), 0xDA7ABA53);
}
/* This test will fail since the expected URL is different to the URL that is
 * passed to connect_to_database() by connect_to_product_database().
 */
void test_connect_to_product_database_bad_url(void **state) {
  expect_string(connect_to_database, url, "products.abcd.com");
  expect_value(connect_to_database, port, 322);
  will_return(connect_to_database, 0xDA7ABA53);
  assert_int_equal((int)connect_to_product_database(), 0xDA7ABA53);
}
```

Mock functions

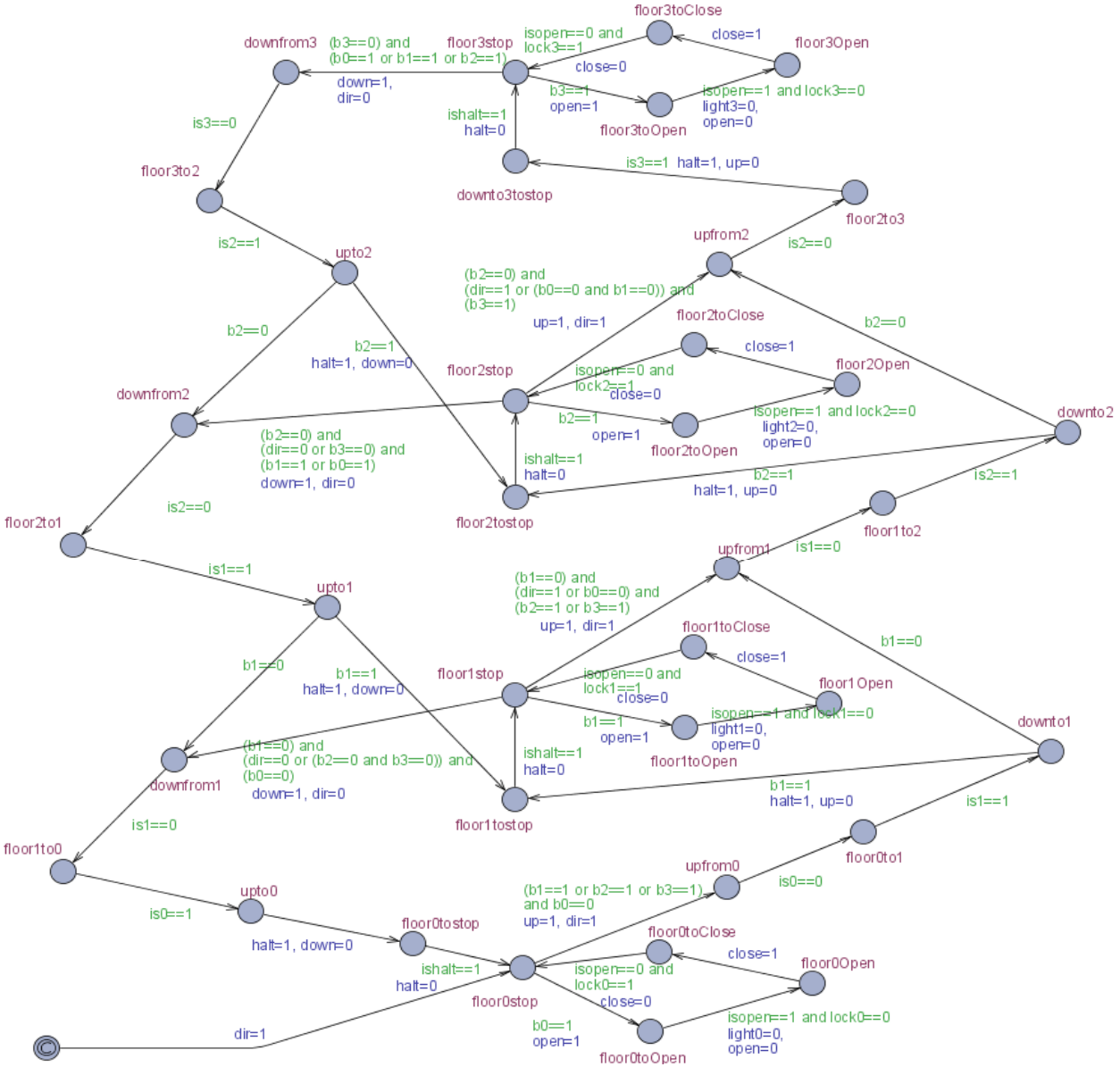
```
/* This test will fail since the mock connect_to_database() will attempt to
 * retrieve a value for the parameter port which isn't specified by this
 * test function.
 */
void test_connect_to_product_database_missing_parameter(void **state)
{
  expect_string(connect_to_database, url, "products.abcd.org");
  will_return(connect_to_database, 0xDA7ABA53);
  assert_int_equal((int)connect_to_product_database(), 0xDA7ABA53);
}

int main(int argc, char* argv[]) {
  const UnitTest tests[] = {
    unit_test(test_connect_to_product_database),
    unit_test(test_connect_to_product_database_bad_url),
    unit_test(test_connect_to_product_database_missing_parameter),
  };
  return run_tests(tests);
}
```

Setup and Teardown (test state)

- Cmockery allows the specification of multiple setup and tear down functions for each test case.
- Setup functions, specified by the unit_test_setup() or unit_test_setup_teardown() macros allow common initialization to be shared between multiple test cases.
- Tear down functions, specified by the unit_test_teardown() or unit_test_setup_teardown() macros provide a code path that is always executed for a test case even when it fails.

An example (elevator)



An example (elevator)